Gremlin Sprite Collision System

Table of Contents

[Overview 2](#_Toc340640701)

[Rendering a scanline 2](#_Toc340640702)

[The ScanBuffer itself 2](#_Toc340640703)

[The ScanBuffer BRAM’s 18bit bus bits 2](#_Toc340640704)

[CPU Accessible Collision Registers 3](#_Toc340640705)

[VideoRegs registers 3](#_Toc340640706)

[BG<N>\_CONTROL 3](#_Toc340640707)

[Collision Control Register (COLCTRLREG) 4](#_Toc340640708)

[Per-sprite Registers (The Collision BRAM) 4](#_Toc340640709)

[Collision Status (COLSTAT) 4](#_Toc340640710)

[Collision Sprite# (COLWITHSPRITEID) 4](#_Toc340640711)

[How collision data is generated 4](#_Toc340640712)

[During Scanline -1 4](#_Toc340640713)

[During every visible scanline 5](#_Toc340640714)

[During BG rendering 5](#_Toc340640715)

[During Sprite rendering 5](#_Toc340640716)

[During Foreground BG rendering 6](#_Toc340640717)

# Overview

The VideoChip renders a scanline to a backbuffer while a frontbuffer is being pushed out the VGA pins.

Tracking collisions, requires remembering what previous renderable object had written a specific pixel into the scanbuffer. This information is fetched at a later time, and compared with the newly rendering item, to evaluate whether a collision has actually occurred.

Sprites can collide with any BG layer, or any other sprite.

There is an optional TEAM mode. In TEAM mode, each sprite can be assigned to either Team 0 or Team 1. In this mode, sprites can only collide with a sprite of the opposing team. I.e. a friendly ship cannot collide with a friendly bullet, but an enemy bullet CAN collide with a friendly ship.

Sprites can be associated with one of two teams. If teams are enabled, then sprites must be on different teams in order to cause a collision between each-other.

In the event of a multi-sprite collision, the sprite that is remembered is the one with the highest SpriteID.

# Rendering a scanline

The order that pixels get rendered on the line, is somewhat random.

**First Pass**: The background color, and all BG’s are drawn into the scanbuffer.

**Second Pass**: All sprites are scanned from 0..255, and each one with a visible row on this scanline, is rendered where it belongs on the scanline.

## The ScanBuffer itself

320 pixels per line (rounded up to 512) x 2 buffers (front and back) = 1024 bytes per line

The ScanBuffer is made from two dual-ported BRAMs configured as a 1K x 36bit bus.

The 1K BRAM address space is broken into two halves: A foreground buffer and a background buffer. The background buffer is where the above rendering is occurring. The foreground buffer is being used to generate the actual VGA signals from the previously rendered scanline.

Channel A: Dedicated to writing pixels into the back buffer, up-to 1 pixel per clock at best.  
Channel B: Dedicated to reading front-buffer pixels, 1 VGA pixel every fourth clock.

### The ScanBuffer BRAM’s Data bus bits

0..3: 4 bits of Blue  
4..7: 4 bits of Green  
8..11: 4 bits of Red  
12..13: 2 bits of Depth  
14..21: 8 bits of Sprite CollisionID  
22: 1 bit of Sprite Present  
23: 1 bit of Sprite on Team (0/1, assuming present)  
24..27: 4 bits of BG Present flags  
  
*NOTE: The above BRAM creates a limitation of 4 bits per channel for the final output image. To get around this, significant redesign of the entire Viode Chip would be required, since all graphic objects and layers are based on 4bit palette entries. I’m ok with this limitation, which still yields 4096 colors.*

*NOTE: The above design is structured to provice all BG rendering entirely in one pass, using the depth buffer and collision data simultaneously. This eliminates the second pass for foreground BG’s that I used to have. It also means that sprites are rendered last, and must therefore obey the per-pixel depth before rendering their pixel(s). Finally, it also simplifies detection of collisions against BG’s that may be on top of (instead of under) a sprite. It even allows a sprite to detect collision against all BG’s perfectly accurately, regardless of visibility, because the CollisionID contains a bitfield of BG presence flags.*

*NOTE: The Depth value is intended for rendering purposes only. The updating of collision data is not affected by Depth. Collision data is affected only by SpriteID and BG presence. Therefore, a pixel in the scanbuffer may have the color and depth value from a BG, but collision data from a sprite. To hide collision data according to depth, would be to make all hidden sprite pixels immune to collision.*

# CPU Accessible Collision Registers

Ideally, I want to make the collision data available at the end of the frame, and retain its’ validity until next time collision data is requested.

There is a bank of memory in the VideoChip, that offers the collision data to the CPU. It is broken into two buffers. One buffer is the frozen buffer, and the other buffer is where the VideoChip is currently generating collision data into.

Since the collision data is generated by the act of rendering the screen, the collision data requires a full VBlank to completely update, and it appears to be incorrect until the entire frame is visited, from start to finish.

On VBLANK, the VideoChip has just completed generating new collision data into the un-frozen collision buffer. The CPU asks to freeze the other collision buffer, before the first scanline is visited. The CPU is now free to make use of the newly frozen collision buffer for as long as it wants. The VideoChip will repeatedly generate new collision data into the un-frozen buffer until asked to swap again.

Note that there is nothing protecting the system from arbitrarily freezing a specific collision buffer whenever it feels like it. However, the content will be corrupted, if the timing is not actually synchronized with the VBLANK, as described above.

## VideoRegs registers

### BG<N>\_CONTROL

Bit 0..1: Depth A new bitfield, in an existing register. Allows the BG to render at any depth relative to other BG’s and sprites.

Bit 2: Foreground **REMOVE -**  This flag is not required anymore, we now have a depth buffer.  
Note that this flag is intended to become part of the Mode select (for PostEffect, 256 color, and RGB modes), in a future version.

Bit 7: Visible The Visible flag is moved to the end of the byte.

### Collision Control Register (COLCTRLREG)

Bit 0 **Freeze buffer #0 or #1**. Set this bit to the desired value, early in the VBLANK ISR. This will freeze the data in the requested buffer so that the CPU can interpret it at its leisure. The next video frame will generate collision data into the un-frozen buffer. This flag takes effect immediately.

Bit 1 **Use Teams**. 1 = A sprite can collide only with sprites of the opposing team. 0 = A sprite can collide with any other sprite.

Bit 2..7 **(not used)**

## Per-sprite Registers (The Collision BRAM)

The following registers are located within the Collision BRAM. The BRAM is organized as 1K x 16, so that there can be two buffers of 512 x 16 each (the frozen, and unfrozen buffers).

There is one set of each of the following registers, for each available sprite.

The CPU should only be looking at the frozen buffer, as the data in the un-frozen buffer is highly unlikely to be valid.

The BRAM is dual-ported. One port is dedicated to the CPU. The other port is dedicated to the ScanBuffer Rendering Engine, which will update the collision data in the un-frozen buffer as each new frame is rendered.

### Collision Status (COLSTAT)

Bit# CPU VideoChip NAME Description  
0..1 R/W R/\_ DEPTH A render-depth for this sprite.  
2 R/W R/\_ TEAM My Team# (0 or 1).  
3 R/\_ \_/W SPR Collided with a sprite (filtered appropriately by team if enabled).  
4 R/\_ \_/W BG0 Collided with BG0.  
5 R/\_ \_/W BG1 Collided with BG1.  
6 R/\_ \_/W BG2 Collided with BG2.  
7 R/\_ \_/W BG3 Collided with BG3.

### Collision Sprite# (COLWITHSPRITEID)

Bit# CPU VideoChip Description  
0..7 R/\_ \_/W The highest SpriteID that this sprite covered, or FF if no collision.

*NOTE: The above value can never contain an FF due to an actual collision, because a sprite must lie ON TOP of another sprite, before a collision is acknowledged. Since sprite 255 is the last one to be rendered, no other sprite will ever lie on top of it!*

# How collision data is generated

## During Scanline -1

The following new actions are performed:

* The unfrozen CollisionBuffer is cleared, indicating no collisions, as follows:
  + Stage 0
    - Cycle 0
      1. Read CollisionBuffer
    - Cycle 1
      1. Write Collision Buffer

Each sprite in the CollisionBuffer, has all VideoChip-writable bits cleared. Additionally, the SpriteID is set to FF. Note that the above, should take 256 reads and 256 writes to complete (512 of the 1280 available clocks on the final VGA scanline before visible stuff starts drawing).

## During every visible scanline

### During BG rendering

The following new actions are performed, per-pixel of the scanbuffer:

* Stage 0
  + Cycle 0
    - Read the cell# from the MapData for all BG’s
* Stage 1
  + Cycle 0
    - Read the color index from the CellData for all BG’s
* Stage 2
  + Cycle 0
    - Establish which BG is the winner. Choose its’ index and palette, or 0,0 if no winner.
    - Read the winning palette entry.
* Stage 3
  + Cycle 0
    - Set the following in the ScanBuffer:
      1. CollisionType is set to either ‘Nothing’ or ‘BG’, depending on whether the pixel is ultimately the background color or not.
      2. CollisionID is set to the mask of BG’s that had a non-transparent pixel, regardless of BG Visibility.
      3. Depth is set to the highest depth value of all BG’s, or if a tie, the depth value of the highest BG involved in the tie.
      4. If not isTransparent
         1. R = isTransparent ? wasR : winningR
         2. G = isTransparent ? wasG : winningG
         3. B = isTransparent ? wasB : winningB

The above can occur at full clock-rate. I .e. One clock per pixel, which is double the pixel throughput vs the old clock-rate. However, there are an extra 2 clocks of pipeline delay, now.

Algorithm to prefer highest BG# with highest depth:  
BG3.depth==3 ? bg3Wins :  
BG2.depth==3 ? bg2Wins :  
BG1.depth==3 ? bg1Wins :  
BG0.depth==3 ? bg0Wins :  
BG3.depth==2 ? bg3Wins :  
BG2.depth==2 ? bg2Wins :  
BG1.depth==2 ? bg1Wins :  
BG0.depth==2 ? bg0Wins :  
BG3.depth==1 ? bg3Wins :  
BG2.depth==1 ? bg2Wins :  
BG1.depth==1 ? bg1Wins :  
BG0.depth==1 ? bg0Wins :  
BG3.depth==0 ? bg3Wins :  
BG2.depth==0 ? bg2Wins :  
BG1.depth==0 ? bg1Wins :  
BG0.depth==0 ? bg0Wins :

### During Sprite rendering

#### When Searching for a Sprite to Render

#### When Rendering the Pixels of a Sprite Scanline

The following events are designed to work as a two-stage pipeline, two cycles per stage.

* Stage 0
  + Cycle 0
    - Read Scanline Buffer at this pixel
      1. Gives us the previous CollisionType
      2. Gives us the previous CollisionID
      3. Gives us the previous Depth
    - (read the sprite pixel index & palette)
  + Cycle 1
    - Read PaletteBuffer (indexed -> RGB)
* Stage 1
  + Cycle 2
    - Read the Sprite Collision Bram
      1. Gives us the Team# on the next clock.
      2. Gives us the Depth to use for the sprite.
      3. Gives us the existing known collisions for this sprite, on the next clock.
  + Cycle 3
    - Write Scanline Buffer at this pixel with the following additional data (assuming no transparency):
      1. CollisionType == Sprite On Team N
      2. CollisionID = This sprite’s ID
      3. Depth = This sprite’s Depth
    - Write the Sprite Collision BRAM
      1. Keep the same Team#
      2. If CollisionType == Sprite && (teams are different || team mode off), then set COLSTAT.SPR = 1, SpriteID = MAX(ScanBuffer.CollisionID, SpriteCollisionReg.CollisionID).
      3. If CollisionType == Bg && CollisionID == 0, then set COLSTAT.BG0 = 1
      4. If CollisionType == Bg && CollisionID == 1, then set COLSTAT.BG1 = 1
      5. If CollisionType == Bg && CollisionID == 2, then set COLSTAT.BG2 = 1
      6. If CollisionType == Bg && CollisionID == 3, then set COLSTAT.BG3 = 1

Note that the above logic requires a pipeline with two stages, and two cycles per stage to complete each sprite pixel, where only one single cycle was needed before adding collision. Overall, since the clock-rate was doubled, this new pipeline is only slower by a grand-total of two clocks per sprite (512 clocks total, assuming all sprites are visible on the scanline).

### During Foreground BG rendering

Foreground BG Rendering has been completely eliminated, thanks to the Depth Buffer! This saves 320 cycles at the end of each pixel scanline.

# Implementation

## Immediate Changes

* [DONE] Convert the Scanbuffer to RGB
* [DONE] Move the Palette Lookup to occur inside the ScanBufferRenderManager

## Future Changes

* The 4 BG memories can be merged in various ways
  + 0+1 map to BG0 in 256 color mode
  + 2+3 map to BG2 in 256 color mode
  + 0+1 map to BG0 in RGB color mode
  + 2+3 map to BG2 in RGB color mode
  + BG3 can act as a mask over the entire screen
    - 0000 = Normal
    - 0001 = Quarter Random
    - 0010 = Half Random
    - 0011 = Full Random
    - 0100 =
    - 0101 =
    - 0110 = Inverse
    - 0111 = Transparent (invisible)
    - 1000 = Black
    - 1001 = ThreeQuarter Darker (RGB – 3)
    - 1010 = Half Darker (RGB - 2)
    - 1011 = Quarter Darker (RGB - 1)
    - 1100 = Quarter Brighter (RGB + 1)
    - 1101 = Half Brighter (RGB + 2)
    - 1110 = ThreeQuarter Brighter (RGB + 3)
    - 1111 = White

# TODO

* [DONE] Implement logic for **regFrozenCollisionDataPage**.
* Implement logic for **regCollisionUsesTeams**.
* [DONE] Implement BGn Depth regs
* [DONE]Moved Visible flag to bit 7
* [DONE] Implement new BG control register format (depth, no foreground, visible, as above)
* Implement Sprite ClipRect